Building REST API with Spring Boot

Sai Upadhyayula

Version 1.0 | 20-01-2023

Building REST API with Spring Boot

Table of Contents

Dedication

This ebook is dedicated to my lovely wife Sowmya and my son Ayansh.

Preface

Who this book is for

This book is for developers who want to build robust REST APIs using Spring Boot. Both beginner and experienced developers can benefit from this book.

What you need for this book

To try code samples in this book, you will need a computer running an up-to-date operating system (Windows, Linux, or Mac OS X). You will need Java 17 and MongoDB installed. The book code was tested against JDK 17, but newer versions should also work.

Conventions

We use a number of typographical conventions within this book that distinguish between different kinds of information.

Code in the text, including commands, variables, file names, CSS class names, and property names are shown as follows:

> Spring Boot uses a `public static void main` entry-point that launches an embedded web server for you.

A block of code is set out as follows. It may be colored, depending on the format in which you're reading this book.

Listing 1. src/main/java/demo/DemoApplication.java

```java
@RestController
class BlogController {
    private final BlogRepository repository;

    // Yay! No annotations needed for constructor injection in Spring 4.3+.
    public BlogController(BlogRepository repository) {
        this.repository = repository;
    }

    @RequestMapping("/blogs")
    Collection<Blog> list() {
        return repository.findAll();
    }
}
```

When we want to draw your attention to certain lines of code, those lines are annotated using numbers accompanied by brief descriptions.

```java
@RestController ①
class BlogController {
    private final BlogRepository repository;

    public BlogController(BlogRepository repository) { ②
        this.repository = repository;
    }

    @GetMapping("/blogs") ③
    Collection<Blog> list() {
        return repository.findAll();
    }
}
```

① The `BlogController` class is annotated with the `@RestController` annotation indicating that it is a Spring Controller which handles the REST Based Requests from the Client.

② We are injecting the `BlogRepository` class into the `BlogController` class through `Constructor Injection`.

③ The `@GetMapping` annotation is responsible to define an HTTP GET Endpoint at the url - `/blogs`

Tips are shown using callouts like this.

Warnings are shown using callouts like this.

Sidebar

Additional information about a certain topic may be displayed in a sidebar like this one.

Finally, this text shows what a quote looks like:

> In the end, it's not the years in your life that count. It's the life in your years.
>
> — Abraham Lincoln

Reader feedback

Let me know what you thought about this book — what you liked or disliked. Reader feedback helps me develop more ebooks and content that deliver the most value to you.

To send us feedback, e-mail us at programmingtechie@gmail.com, send a tweet to @sai90_u (https://twitter.com/sai90_u).

Introduction

Spring Boot has literally revolutionized the way Java applications are built.

It’s a widely used Java Framework in the industry and helps in rapid development of Web, REST based applications. In the modern software development world, microservices became very popular and as many of these microservices are built as RESTful APIs, it’s essential to understand and learn how to build robust REST APIs.

In this ebook, you will learn how to "Build a REST API with Spring Boot".

This is a beginner level book, we will start with fundamentals of Spring Boot, fundamentals of REST API, and gradually we will dive into advanced REST API concepts and learn how to implement them using Spring Boot.

What are you going to build ?

I always believe in the approach "Learn by building". So instead of simply using individual code snippets and examples, as the title suggests, we are going to build a REST API for an `Expense Tracker` application where users can track their expenses.

Technologies used in the Expense Tracker application

The `Expense Tracker` application is built using the following technologies:

- Spring Boot
- MongoDB
- Open API 3.0 for REST API Documentation
- Junit5 for writing Unit Tests
- Testcontainers for Integration Tests

Source code

You can find the source code for this ebook in Github at - https://github.com/SaiUpadhyayula/expense-tracker-rest-api

In the next chapter, we will start our journey by learning the fundamentals of Spring Boot.

PART ONE

What is Spring Boot ?

Spring Boot is an opinionated Java Framework which is used to build robust web applications and REST APIs.

Spring Boot was developed to make it easy to build applications using `Spring Framework`, which started off as a Dependency Injection Framework but grew into much bigger than that by spawning many other projects like :

- Spring MVC - used to build MVC based web applications and RESTful web services
- Spring Security - used to secure the applications built on top of Spring Framework
- Spring Cloud - used to develop distributed systems by implementing commonly used design patterns like Service Discovery, Configuration Management, API Gateway, Circuit Breaker, Observability, etc.
- Spring Integration - used to provide various support for Enterprise Integration Patterns like SFTP, AMQP, JMS, etc.
- Spring WebFlux - used to develop web applications using the Reactive Programming paradigm by building on top of Project Reactor.

There are many more projects under the Spring Project Umbrella - you can track all those projects at the link - https://spring.io/projects

Features of Spring Boot

Spring Boot helps us in rapid development of the applications by providing various features like:

- Embedded Web Servers like Tomcat, Jetty and Undertow
- Providing various `starter` libraries and configuring them automatically using the feature called as "Auto Configuration".

and many more.

You can refer to Spring Boot Reference Documentation to learn more about Spring Boot - https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

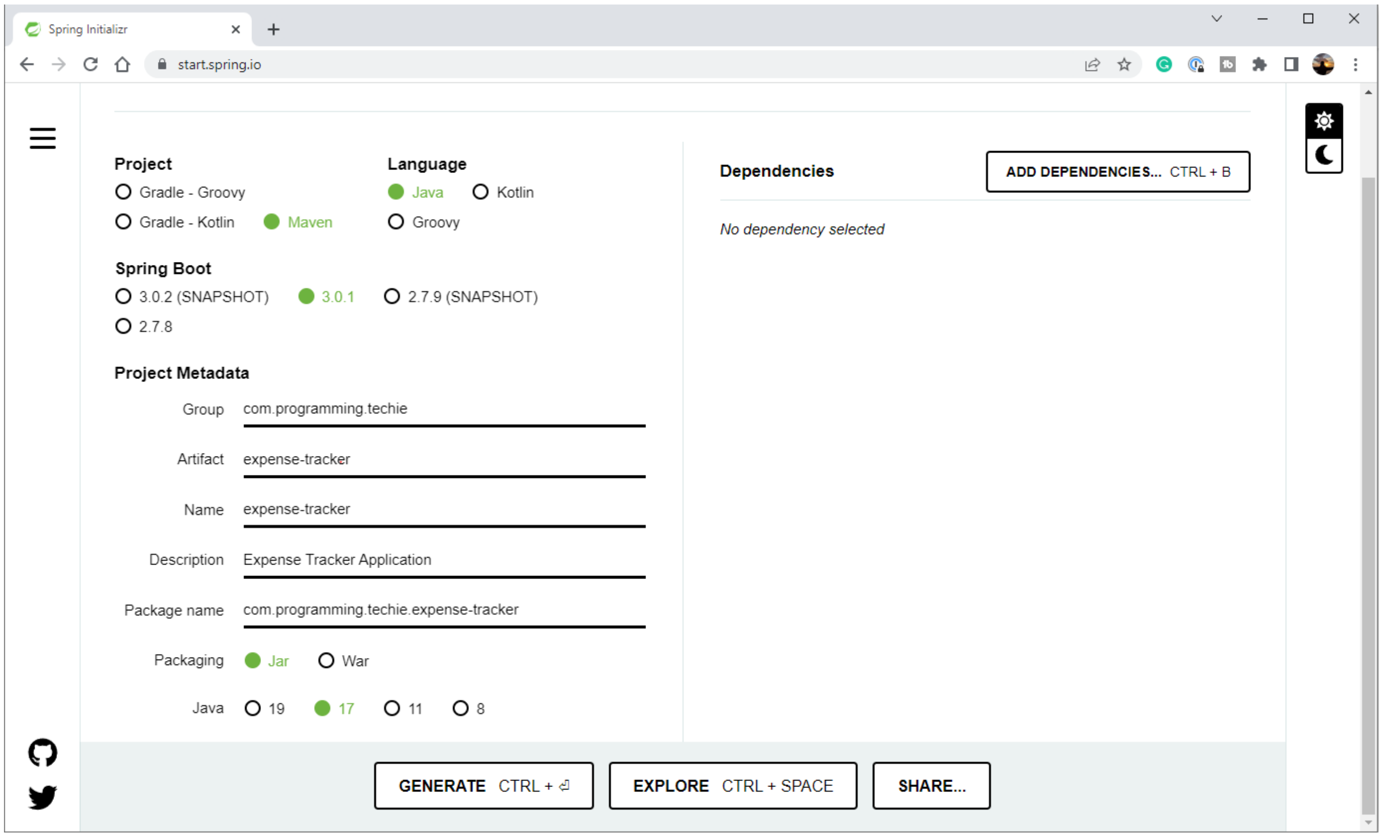
In the next chapter, we will start by building your First Spring Boot Application.

PART TWO

Building your First Spring Boot Application

Generating the Starter Project

To get started with building Spring Boot applications, you can visit the Spring Initializer website at https://start.spring.io/



Project: Gradle - Groovy, Gradle - Kotlin, [✓ Maven]

Language: [✓ Java], Kotlin, Groovy

Spring Boot: 3.0.2 (SNAPSHOT), [✓ 3.0.1], 2.7.9 (SNAPSHOT), 2.7.8

Project Metadata

Group	com.programming.techie
Artifact	expense-tracker
Name	expense-tracker
Description	Expense Tracker Application
Package name	com.programming.techie.expense-tracker
Packaging	[✓ Jar] War
Java	19 [✓ 17] 11 8

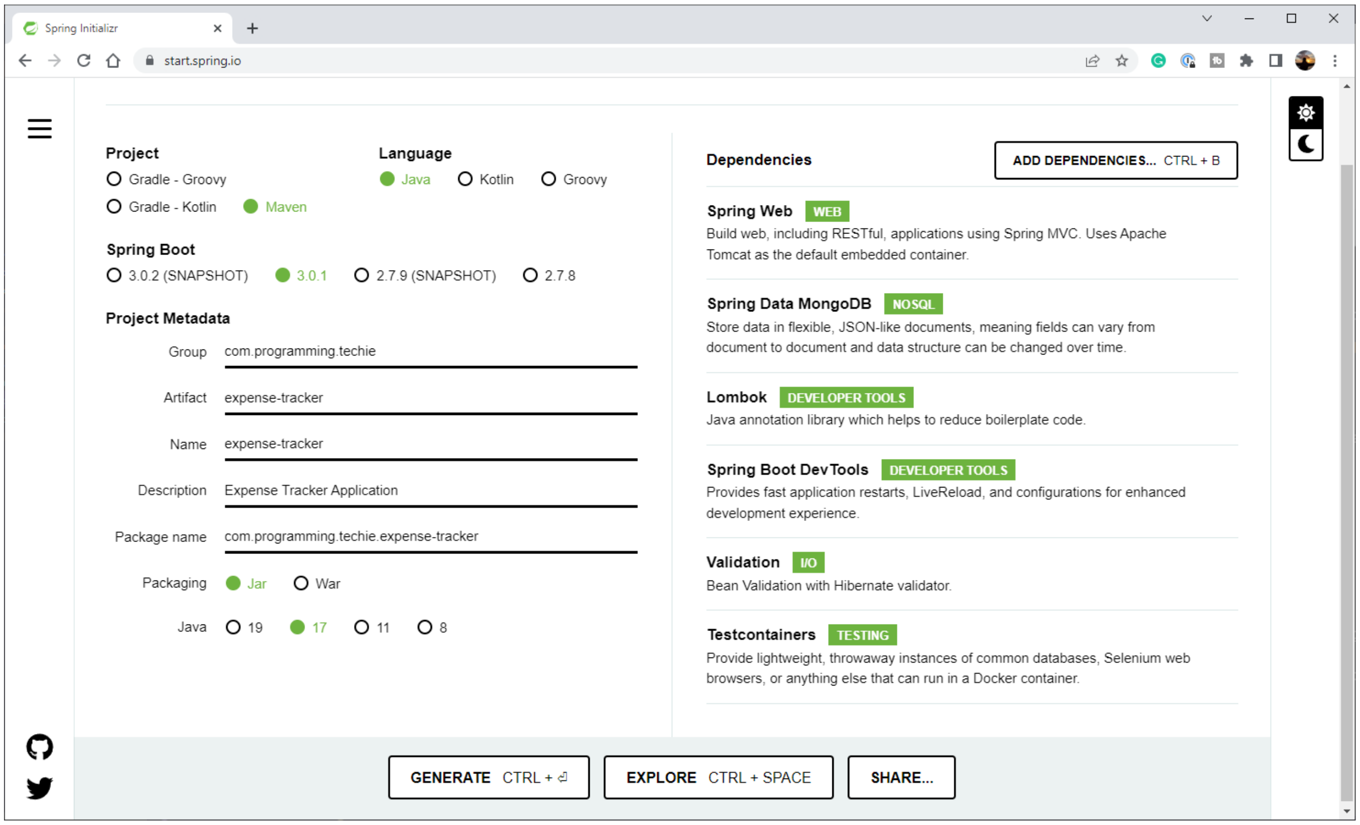
Dependencies

No dependency selected

In the website, select the following options:

- **Project**: Maven (You can also use Gradle if you are comfortable with it)
- **Spring Boot**: We are going to use the latest version - 3.0.1 at the time of writing this book.
- **Language**: Java
- **Project Metadata**: (You can provide any values as you like)
  - **Group**: com.programming.techie
  - **Artifact**: expense-tracker
  - **Name**: expense-tracker
  - **Description**: Expense Tracker Application
  - **Packaging**: Jar
  - **Java**: 17 (The latest LTS version at the time of writing this book)

After adding the project metadata, now it's time to select the dependencies we are going to use in our `Expense Tracker` application.



Project: ○ Gradle - Groovy ○ Gradle - Kotlin [✓ Maven]

Language: [✓ Java] ○ Kotlin ○ Groovy

Spring Boot: ○ 3.0.2 (SNAPSHOT) [✓ 3.0.1] ○ 2.7.9 (SNAPSHOT) ○ 2.7.8

Project Metadata

Group	com.programming.techie
Artifact	expense-tracker
Name	expense-tracker
Description	Expense Tracker Application
Package name	com.programming.techie.expense-tracker
Packaging	[✓ Jar] ○ War
Java	○ 19 [✓ 17] ○ 11 ○ 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data MongoDB NOSQL
Store data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Validation I/O
Bean Validation with Hibernate validator.

Testcontainers TESTING
Provide lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.

- **Dependencies**:
  - **Spring Web** - adds support to develop MVC based web applications as well as RESTful applications.
  - **Spring Data MongoDB** - Adds support for integrating with the MongoDB.
  - **Lombok** - Adds support to Java annotation Library which reduces the boilerplate we have to write in Java
  - **Spring Boot Dev Tools** - Adds support for LiveReload
  - **Validation** - Adds the Hibernate Validator Support
  - **Testcontainers** - Provides lightweight throwaway instances of common databases, message queues, Selenium Web browers, etc.

After choosing all the above-mentioned options, you can go ahead and click on the `Generate` button to download the starter code as a Zip file.

You can also view the generated starter code by clicking on the `Explore` button, this will open a popup showing all the files you are about to download.

Once you have downloaded the starter code, you can open them in your favourite IDE like VS Code, Intellij IDEA, Eclipse, Netbeans (or) whatever IDE you are using ;)

Exploring the Project Structure

If you extract the zip file, you can find the following Project Structure for the Spring Boot Application

```
|src
|-- main
    |-- java
    |-- resources
|-- test
    |-- java
| pom.xml
```

This is the standard project structure for any Java project based, if you are using Maven as your build tool, then you will have a pom.xml file inside the root folder of the project.

This `pom.xml` contains all the metadata of the project like `groupId`, `artifactId`, `version`, `dependency`, `plugins` etc.

This is how the `pom.xml` looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent> ①
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.1</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.programming.techie</groupId>
  <artifactId>expense-tracker</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>expense-tracker</name>
  <description>Expense Tracker Application</description>
  <properties>
    <java.version>17</java.version>
    <testcontainers.version>1.17.6</testcontainers.version>
  </properties>
  <dependencies> ②
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>junit-jupiter</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>mongodb</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement> ③
        <dependencies>
            <dependency>
                <groupId>org.testcontainers</groupId>
                <artifactId>testcontainers-bom</artifactId>
                <version>${testcontainers.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
```

```
      </dependencies>
    </dependencyManagement>

    <build>
      <plugins> ④
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
          <configuration>
            <excludes>
              <exclude>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
              </exclude>
            </excludes>
          </configuration>
        </plugin>
      </plugins>
    </build>

</project>
```

① The first thing you can observe inside the `pom.xml` file is the `<parent>` tag which defines a parent for this project. The `spring-boot-starter-parent` project is defined as a default parent for all the Spring Boot Projects ,and this project contains all the necessary libraries to run the project as a Spring Boot Application.

② The `<dependencies>` tag helps us to define the libraries we want to include in our project as individual `<dependency>`, here you can observe that we have some starter libraries like `spring-boot-starter-data-mongodb`, `spring-boot-starter-validation`, `spring-boot-starter-web` which brings in the necessary functionalities to our Spring Boot project.

③ The `<dependencyManagement>` is generally used to manage the version of the dependencies in the projects, have a look at the next section to understand more about this.

④ We have also some plugins defined under the `<plugins>` section, the `spring-boot-maven-plugin` is used to create a JAR file for our application.

Dependency Management in Maven Projects

Let's understand why we need the `<dependencyManagement>` tag and how it works. Imagine that we have the following Maven Project Structure:

```
Parent Maven Project
  - Child Maven Project 1
  - Child Maven Project 2
  - Child Maven Proejct 3
```

Now in these 3 Child Maven Projects, if you are using the following dependency:

```
    <dependency>
      <groupId>org.testcontainers</groupId>
      <artifactId>junit-jupiter</artifactId>
      <scope>test</scope>
      <version>1.17.6</version>
    </dependency>
```

Then it's hard to maintain a single version of this dependency (1.17.6) across all the child projects, if a developer decides to change this version of the dependency in a child project for some reason, then we have potentially different version of the same dependency, which can cause weird issues.

For this reason a good practice is to create a Parent Project usually called as a BOM (Bills of Material) where you define the required dependencies and their version.

And then, inside the child project, add the parent project bom under the `<dependencyManagement>` section. Now all you need to do is to define the above mentioned dependency without using the `<version>` tag, and Maven will automatically read the `<version>` from the project mentioned inside the `<dependencyMangement>` section.

For this reason, you can observe that the version for `junit-jupiter` dependency is not defined inside our `pom.xml` file. The version of this dependency is managed by the `<testcontainers-bom>` project.

You can view the contents of the dependencies like `spring-boot-starter-web`, `spring-boot-starter-parent` by pressing the `Ctrl` button and clicking on the dependency name, many modern IDE's provide the functionality to download the required dependencies and opens the `pom.xml` file automatically.

Exploring the SpringBootApplication main class

Under the src→main→java→<your-package>, you should find the primary class which is responsible to start our Spring Boot Application, in our project this is named as `ExpenseTrackerApplication`.

This is how our `ExpenseTrackerApplication ` looks like :

```java
package com.programming.techie.expensetracker;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication ①
public class ExpenseTrackerApplication {

    public static void main(String[] args) { ②
        SpringApplication.run(ExpenseTrackerApplication.class, args);
    }

}
```

① The first thing you can observe is the `@SpringBootApplication` annotation, which is the main annotation responsible to define our application as a Spring Boot application.

② Inside the main method, we have the `SpringApplication.run(..)` which is responsible to take the input arguments and launch our application.

The `@SpringBootApplication` annotation does a lot of things for us in the background, it uses a mechanism called `AutoConfiguration` to scan all the libraries defined inside our `pom.xml` (or) `build.gradle` (equivalent of `pom.xml` when using Gradle) and automatically configure the required objects to use them in our application.

For example, when we define the `spring-boot-starter-web` inside the pom.xml file, Spring Boot will automatically create objects (also called as `Beans`) to create an Embedded Tomcat container, to enable JSON support in our application, etc.

Prior to Spring Boot, we used to define all these beans manually which used to be a lot of overhead while developing applications.

This is how the `@SpringBootApplication` annotation looks like if you inspect it:

```
@SpringBootApplication annotation , which is a composed annotation.
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
                @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
                @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
    ....
    ....
}
```

`@EnableAutoConfiguration` is the annotation which is responsible to add the AutoConfiguration capabilities to our Spring Boot Application.

Now if you run the `ExpenseTrackerApplication` class in your IDE, you can observe the following output:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.1)

2023-01-20T16:53:41.595+01:00  INFO 21860 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : Starting ExpenseTrackerRestApiApplication
using Java 17.0.1 with PID 21860 (F:\playground\expense-tracker-rest-api\target\classes
started by subra in F:\playground\expense-tracker-rest-api)
2023-01-20T16:53:41.599+01:00  INFO 21860 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : No active profile set, falling back to 1
default profile: "default"
2023-01-20T16:53:42.023+01:00  INFO 21860 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data MongoDB repositories
in DEFAULT mode.
2023-01-20T16:53:42.034+01:00  INFO 21860 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 8
ms. Found 0 MongoDB repository interfaces.
2023-01-20T16:53:42.299+01:00  INFO 21860 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
......
......
......
2023-01-20T16:53:43.197+01:00  INFO 21860 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
2023-01-20T16:53:43.204+01:00  INFO 21860 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : Started ExpenseTrackerRestApiApplication in
1.863 seconds (process running for 2.304)
```

You can observe that an Embedded Tomcat Webserver is started automatically at the default port - 8080.

You can change the default port from 8080 to any arbitary port by adding the following property under the `src/main/resources/application.properties` file.

```
server.port=8081
```

If you restart the application after adding the above property, you can observe that our application now start at port - 8081.

```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.1)

2023-01-20T17:01:36.347+01:00  INFO 26396 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : Starting ExpenseTrackerRestApiApplication
using Java 17.0.1 with PID 26396 (F:\playground\expense-tracker-rest-api\target\classes
started by subra in F:\playground\expense-tracker-rest-api)
2023-01-20T17:01:36.351+01:00  INFO 26396 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : No active profile set, falling back to 1
default profile: "default"
2023-01-20T17:01:36.770+01:00  INFO 26396 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data MongoDB repositories
in DEFAULT mode.
2023-01-20T17:01:36.786+01:00  INFO 26396 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 7
ms. Found 0 MongoDB repository interfaces.
2023-01-20T17:01:37.059+01:00  INFO 26396 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8081 (http)
2023-01-20T17:01:37.073+01:00  INFO 26396 --- [           main]
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-01-20T17:01:37.073+01:00  INFO 26396 --- [           main]
o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.4]
2023-01-20T17:01:37.141+01:00  INFO 26396 --- [           main]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded
WebApplicationContext
2023-01-20T17:01:37.141+01:00  INFO 26396 --- [           main]
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization
completed in 756 ms
....
....
....
2023-01-20T17:01:37.976+01:00  INFO 26396 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8081 (http) with
context path ''
2023-01-20T17:01:37.982+01:00  INFO 26396 --- [           main]
c.p.t.e.ExpenseTrackerRestApiApplication : Started ExpenseTrackerRestApiApplication in
1.91 seconds (process running for 2.373)
```

The `application.properties` is the place where you can define different properties to configure your Spring Boot Application.

For example, you can define the database connection URL and credential information under the `application.properties` file, also if you want to define a custom property like an external URL, then you can define them inside this file.

Instead of using a properties file, you can also define a yml file if you are more inclined to write YAML. Then it should be defined as `application.yml` instead of `application.properties`

Summary

In this chapter you learned how to generate a Spring Boot Project using the Spring Initializr website, and then we explore the package structure, understood how the dependencies are managed inside the `pom.xml` file. We understood how a basic Spring Boot application looks like and configured and finally we ran the Spring Boot Application.

In the next section, we are going to start understanding the fundamentals of REST and start creating the REST API using Spring Boot.

PART THREE

Understanding REST Fundamentals

REST stands for REpresentational State Transfer, this term was coined by Roy Fielding.

REST is an architectural style for building distributed systems that provide interoperability between heterogeneous systems.

It defines a way of interaction between different kinds of client in a uniform way.

In the recent times there are multiple clients like Mobile Devices, Single Page Applications, Desktop Applications, IOT Devices which consume data from the backend. Instead of developing standalone services for each client, we can develop an API which follows RESTful conventions and use that API for all the clients which also consumes this API using the RESTful conventions.

You can learn more about REST at this link - https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

A key term comes up whenever we are talking about a RESTful API, and that is called as a `Resource`.

A `Resource` can be identified as any information which is exposed to the Web and which can be referred using a unique identifier.

Our Expense Tracker application is used to keep track and maintain the Expenses, so in our case an Expense can be defined as a Resource, and the URL to access that resource can be identified as a Unique Identifier. For example, we can define a URL to read a particular expense like below:

```
http://localhost:8080/api/expense/1
```

HTTP Verbs

We can denote any action performed on a `Resource` in the form of HTTP Verbs like below:

- GET - Operation to Get (or) Read a Resource
- POST - Operation to Create a Resource
- PUT - Operation to Update a Resource
- PATCH - Operation to Update a Part of a Resource instead of whole Resource.
- DELETE - Operation to Delete a Resource

These are the most widely used HTTP Verbs in the world of RESTful APIs.

URL Representation of HTTP Verbs

Now let's see how a standard URL looks like for each of the above-mentioned HTTP Verbs, which can be identified as a HTTP Method:

HTTP Verb	URI	Action
GET	http://localhost:8080/api/expense/{expenseId}	Get Expense with HTTP Status 200
POST	http://localhost:8080/api/expense	Creates Expense specified inside Request Body, return HTTP Status 201 and the URI of the created `Resource` in the `Location` Header
PUT	http://localhost:8080/api/expense/{expenseId}	Update Expense specified inside Request Body, return HTTP Status 200
PATCH	http://localhost:8080/api/expense/{expenseId}	Partially Update Expense specified inside Request Body, return HTTP Status 200
DELETE	http://localhost:8080/api/expense	Delete Expense specified inside Request Body, return HTTP Status 200

REST URL Naming Conventions

There are some pre-defined conventions which we have to follow to define the URL of the REST API.

Use Nouns instead of verbs

As we know, APIs are always designed around Resources, so in our example it's going to be the `Expense` class.

Any kind of operations we are going to perform against those resources should be defined through the HTTP Verbs like (**GET**, **POST**, **PUT**, **DELETE**)

Instead of specifying the action inside the URL, it's a good practice to just use the name of the Resource instead of specifying the action.

For example:

If you want to define the URL for the Add Expense Endpoint, instead of naming the URL as `/api/expense/add` we are going to name it as `/api/expense/` and the action is taken care by the corresponding HTTP Verb we will use, in this case it's going to be **POST**.

Use Lowercase Letters

Lowercase Letters should be preferred for the URI's as much as possible. According to [RFC-3986](https://www.ietf.org/rfc/rfc3986.txt) URIs are case-sensitive except for the scheme(http/https) and Host components (hostname of the server).

Do not use file extensions

If you want to access a file, never use the extension as part of the URI, this is because URI should be independent of the implementation.

For example instead of defining the URL for a pdf file like below:

`/api/expense/report.pdf`

Define it like below:

`/api/expense/report`

In the above case, we will communicate our intent that we need a PDF file through the HTTP `Content-Type` header, by using the **Media Type** parameter in the HTTP Response and by using the **Accept** Header in the request.

You can find much more API Guidelines like these in this link - https://opensource.zalando.com/restful-api-guidelines/#table-of-contents

Please note that these are Guidelines defined by Zalando according to their requirements, take this as opinionated set of guidelines, but not as strict rules to follow.

Content Negotiation by RESTful Webservices

RESTful Web Services can support both XML (or) JSON payloads as the Request Body.

But the majority of the modern services communicate with RESTful backends in the form of JSON (Javascript Object Notation)

Summary

In this chapter, we learned the basics of REST and understood the different terminogies associated with REST.

We learned some best practices involved in representing the Resource URL and learned what are the different Request Body types supported by REST services.

In the next chapter, we will start with the implementation of our Expense Tracker Application.

PART FOUR

Building REST API using Spring Boot

Let’s go ahead and start developing our Expense Tracker REST API.

We will start by creating the domain model of our Expense Tracker Application.

Domain Model

An Expense Tracker Application can have the following domain classes:

- Expense
- ExpenseCategory
- User

Here is how we can model the `Expense.java` class:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class Expense {
    private String id;
    private String expenseName;
    private ExpenseCategory expenseCategory;
    private BigDecimal expenseAmount;
}
```

This is a simple Plain Old Java Object (POJO) class, but with a twist, you can observe that we don’t have usual blocks of Java code like No-Args and Argument Constructor, Getter/Setter, etc.

These are all generated at compile-time using the library we added in chapter-2 called as `lombok` which will help us by generating the Java Boiler Plate code.

You can observe the annotations `@Getter`, `@Setter`, `@AllArgsConstructor`, `@NoArgsConstructor` they are pretty self-explanatory.

`@Builder` annotation generates the necessary code to implement the `Builder` design pattern. You can learn more about `Builder Design Pattern` here - https://refactoring.guru/design-patterns/builder

The `Expense.java` class defines a field of type `ExpenseCategory` which is a Java Enum that looks like below:

```
public enum ExpenseCategory {
    ENTERTAINMENT, GROCERIES, RESTAURANT, UTILITIES, MISC
}
```

Finally, we have the `User.java` class which looks like below:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class User {
    private String id;
    private String userName;
    private String password;
    private String email;
}
```

After defining the domain model, we can now start creating the REST Endpoints.

Installing MongoDB

As mentioned we will be using MongoDB as our database, you can install MongoDB either by using Docker (or) manually.

You can find the resources to download and install MongoDB Community Edition here - https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/

Configuring MongoDB in Expense Tracker Application

After installing MongoDB, the next step is to configure our Expense Tracker Application with MongoDB.

We can do that by adding the following properties to the `application.properties` (or) `application.yml` file.

```
spring.data.mongodb.uri=mongodb://localhost:27017/expense-tracker
```

The `spring.data.mongodb.uri` contains all the necessary information required to configure the MongoDB in our application.

MongoDB by default runs on port - 27017, if you chose to install it in different port, make sure to use the same port in the URI.

Creating Repositories using Spring Data

The next step is to create the logic to define the classes which are responsible to store the information in the MongoDB.

Spring Boot has a project called as `Spring Data`, that abstracts away the logic to write the usual database operations code.

It implements the famous Repository Pattern, which you can learn more about in this article - https://martinfowler.com/eaaCatalog/repository.html

By using `Spring Data` all you have to do is create an interface called as `ExpenseRepository.java` and extend the interface with `MongoRepository.java`.

This is how the code for `ExpenseRepository.java` looks like:

```
package com.programming.techie.expensetracker.repository;

import com.programming.techie.expensetracker.model.Expense;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.Optional;


public interface ExpenseRepository extends MongoRepository<Expense, String> {
}
```

We have an interface called `ExpenseRepository` which extends the `MongoRepository.java` interface which brings in a lot of functionality to our `ExpenseRepository.java`

Now let’s learn a bit more about the `MongoRepository` interface.

```
package org.springframework.data.mongodb.repository;

import java.util.List;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
import org.springframework.data.repository.ListCrudRepository;
import org.springframework.data.repository.ListPagingAndSortingRepository;
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.query.QueryByExampleExecutor;

@NoRepositoryBean
public interface MongoRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
    <S extends T> S insert(S entity);

    <S extends T> List<S> insert(Iterable<S> entities);

    <S extends T> List<S> findAll(Example<S> example);

    <S extends T> List<S> findAll(Example<S> example, Sort sort);
}
```

The `ExpenseRepository` interface inherits all the above-mentioned methods you see in the `MongoRepository.java` class, and `Spring Data` will provide implementations to this method at run-time.

So you need not worry about opening a Database connection, writing the logic to perform necessary database operations and finally closing the Database Connection. Everything is handled for us by `Spring Data`. Pretty cool, right ?

The next step is to map our Domain Model classes to the Database, we can do that by adding the `@Document` annotation on top of our `Expense.java` class, as we are going to store the expense in the database.

In the traditional Relational Databses, we usually store the data inside the tables. In MongoDB, we do not use Tables, but store them in the form of Documents. That's why we have the `@Document` annotation to represent our Expense class as a MongoDB document.

Here is how the `Expense.java` class looks like after adding the necessary annotations:

```java
package com.programming.techie.expensetracker.model;

import lombok.*;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

import java.math.BigDecimal;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Document("expense") ①
public class Expense {
    @Id ②
    private String id;
    @Field("name") ③
    @Indexed(unique = true) ④
    private String expenseName;
    @Field("category")
    private ExpenseCategory expenseCategory;
    @Field("amount")
    private BigDecimal expenseAmount;
}
```

① As mentioned in the NOTE section above, we add the `@Document` to denote the `Expense` object as a MongoDB Document.

② We define a primary key in our document, by using the `@Id` annotation, notice that the ID created by MongoDB is in the form of Binary Object Notation, you can read more about it here - https://www.mongodb.com/basics/bson

③ We can map each field inside our `Expense` class into the Mongodb document, with the help of the `@Field` annotation. For example, you can have the expense saved with fieldname - "category" instead of "expenseCategory".

Alright, now we have the logic to store the expenses, but how do we receive the expenses from the clients into our application ? Who is responsible for this ? The answer is Controllers, let’s create them in the next section.

Creating Controllers

To start building the REST Endpoints, you need to get familiarized with the term called as `Controller`.

Before that let’s revisit the MVC Design Pattern

```
                         ___________________
                        |                   |
                        |        Model      | ②
                         ___________________
                      /
                     /
 ___________________/
|                   |
|   Controller      | ①
 ___________________
                     \
                      \
                       \
                         ___________________
                        |                   |
                        |        View       | ③
                         ___________________
```

In the MVC Pattern, we have the main components - Model, View and Controllers.

① The Controller is responsible to accept the requests coming from the external client and forwarding it to the application service layer.

② The Model contains the information which is exposed by the webservice/application.

③ The View contains the response usually returned as response by the Controller. In a traditional web application, this can be an HTML page, but in RESTful applications, this can be just a JSON/XML payload as response.

@RestController

Spring Boot provides the functionality of the Controller in the MVC Design Pattern using the `@Controller` annotation, if you want to handle RESTful applications, then it has another annotation called `@RestController` which is nothing but a wrapper around the `@Controller` annotaion.

By adding this annotation, Spring Boot will start listening to incoming requests to the application, and will start responding in the form of either JSON/XML.

Let’s create a class called as `ExpenseController.java`

```
package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.exception.ExpenseNotFoundException;
import com.programming.techie.expensetracker.model.Expense;
import com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;
import java.util.List;

@RestController ①
@RequestMapping("/api/expense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

    @PostMapping ②
    public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
        return ResponseEntity.created(location)
                .build(); ③
    }

    @PutMapping ⑤
    @ResponseStatus(HttpStatus.OK)
    public void updateExpense(@RequestBody ExpenseDto expense) {
        expenseService.updateExpense(expense);
    }

    @GetMapping ④
    @ResponseStatus(HttpStatus.OK)
    public List<ExpenseDto> getAllExpenses() {
        return expenseService.getAllExpenses();
    }

    @GetMapping("/{id}") ⑥
    @ResponseStatus(HttpStatus.OK)
    public ExpenseDto getExpense(@PathVariable String id) {
        return expenseService.getExpense(id);
    }

    @DeleteMapping("/{id}") ⑦
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteExpense(@PathVariable String id) {
        expenseService.deleteExpense(id);
    }

}
```

① We annotate our class with `@RestController` and `@RequestMapping` annotations, so our `ExpenseController` can listen to the incoming requests starting with the path - `/api/expense`.

② We can use the `@PostMapping` annotation to define the addExpense method as a POST Endpoint. You can also observe that we are using a `@RequestBody` annotation to denote that we are expecting a Body inside the HTTP Post Request. When the `@RequestBody` annotation is present, Spring Boot will automatically parse the incoming HTTP Request Body to the ExpenseDto object.

③ Inside the addExpense Method, we are processing the request, by passing the `expenseDto` object to the `addExpense` method of the `ExpenseService.java` class, after that we are creating a URI with the expenseId as the path-variable, and sending that URI as part of the `Location` header in the response. This is the standard way of implementing a response in RESTful applications, whenever you are creating a Resource.

④ We have the `@GetMapping` annotation to map the `getAllExpenses` which returns all the expenses we have in our application. Please note that we can use the `@ResponseStatus` annotation to define the HTTP Response Status we need to send back to the client.

⑤ We have the `@PutMapping` annotation which listens for HTTP PUT requests and executes the method `updateExpense` in the `ExpenseService` class.

⑥ We also have another `@GetMapping` annotation mapped to the `getExpenseByName` method, which as the name suggests retrieves a given expense by its name. Spring Boot provides the `@PathVariable` annotation to read the `name` from the URI and map it to the String variable.

⑦ Finally, we have a `@DeleteMapping` annotation mapped to the `deleteExpense` method, which takes in the `id` of the given expense which is again parsed using the `@PathVariable` annotation and passed to the `deleteExpense` method of the `ExpenseService`.

You can observe that we are using `@RequiredArgsConstructor` annotation which will generate the constructor at compile time with all the required arguments.

Let’s see how the `ExpenseDto.java` object looks like:

```
package com.programming.techie.expensetracker.dto;

import com.programming.techie.expensetracker.model.ExpenseCategory;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExpenseDto {
    private String id;
    private String expenseName;
    private ExpenseCategory expenseCategory;
    private BigDecimal expenseAmount;
}
```

`ExpenseDto.java` acts as a Data Transfer Object (DTO), whose responsibility is to transfer the data between the client and the server.

Why note use Expense class instead of ExpenseDto ?

You may ask why we have to create a seperate class called `ExpenseDto` to model the Request Body, when we are using almost same fields as `Expense`. Why not use `Expense` object itself ?

That's a very valid question.

There are 2 main reasons for using a Data Transfer Object:

1. It's usually not advisable to expose your domain model to the external world, as it usually contains a lot of information which is not relevant to the clients.
2. APIs evolve overtime and needs to handle different kinds of data which are not part of the Domain Model. For example, in the future we may want to add a different field in the Request Body, but we don't need to store it in the database. In that case, we can just add the field to the `ExpenseDto` and need not touch the `Expense` object.

Creating ExpenseService

Now let's also create a class called `ExpenseService.java` which is responsible to handle the actual business logic to perform CRUD operations for the Expense :

```
package com.programming.techie.expensetracker.service;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.exception.ExpenseNotFoundException;
import com.programming.techie.expensetracker.model.Expense;
import com.programming.techie.expensetracker.repository.ExpenseRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.server.ResponseStatusException;

import java.util.List;
import java.util.stream.Collectors;

@Service ①
@RequiredArgsConstructor
public class ExpenseService {

    private final ExpenseRepository expenseRepository;

    public String addExpense(ExpenseDto expenseDto) { ②
        Expense expense = mapFromDto(expenseDto);
        return expenseRepository.insert(expense).getId();
    }

    public void updateExpense(ExpenseDto expenseDto) { ③
        Expense expense = mapFromDto(expenseDto);
        Expense savedExpense = expenseRepository.findById(expenseDto.getId()).
orElseThrow(() -> new ResponseStatusException(HttpStatus.BAD_REQUEST,
                String.format("Cannot Find Expense by ID %s", expense.getId())));
        savedExpense.setExpenseName(expense.getExpenseName());
        savedExpense.setExpenseCategory(expense.getExpenseCategory());
        savedExpense.setExpenseAmount(expense.getExpenseAmount());

        expenseRepository.save(savedExpense);
    }

    public ExpenseDto getExpense(String id) { ④
        Expense expense = expenseRepository.findById(id)
                .orElseThrow(() -> new ExpenseNotFoundException(String.format("Cannot
Find Expense by ID - %s", id)));
        return mapToDto(expense);
    }
    public List<ExpenseDto> getAllExpenses() { ⑤
        return expenseRepository.findAll()
                .stream()
                .map(this::mapToDto).collect(Collectors.toList());
    }

    public void deleteExpense(String id) { ⑥
        expenseRepository.deleteById(id);
    }

    private ExpenseDto mapToDto(Expense expense) {
        return ExpenseDto.builder()
                .id(expense.getId())
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
    }

    private Expense mapFromDto(ExpenseDto expense) {
        return Expense.builder()
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
    }
}
```

① The `ExpenseService` class is annotated with `@Service` annotation which denotes that this class is supposed to be a Service, Spring Boot will then create an object (Bean) for this class during the application start-up.

② The `addExpense` method takes an `ExpenseDto` class and maps the `ExpenseDto` object to an `Expense` object, which is passed to the `insert` method of the `ExpenseRepository` Spring Data Repository Interface.

③ We have the `updateExpense` which takes in an `ExpenseDto` object similar to `addExpense`, maps it to `Expense` object and saves the updated `Expense` object back to the database using the `ExpenseRepository.save()` method.

④ Next, we have the `getExpense` method which retrieves the `Expense` based on the id, using the `findById` method in the `ExpenseRepository`, inherited from the `MongoRepository` method. If the expense is not found with a given id, we throw an `ExpenseNotFoundException`

⑤ The `getAllExpenses` method, retrieves all the expenses present in our database and maps the response to `ExpenseDto` objects, before returning it back in the form of a `List`.

⑥ Lastly, we have the `deleteExpense` method which takes in an `id` of the `Expense` object, and deletes the

Expense.

Use findAll() method judiciously in your production apps, this will read all the documents in the database, if you are dealing with a pretty big database, then your database will take a lot of time to retrieve all the records.

Lastly, here is how our ExpenseNotFoundException class looks like:

```java
package com.programming.techie.expensetracker.exception;

public class ExpenseNotFoundException extends RuntimeException {
    public ExpenseNotFoundException(String message) {
        super(message);
    }
}
```

Implement Validation for our REST API

So we built the basic REST API, but we can improve this API by adding input validation.

As of now, we can send all kinds of invaild input to our REST API.

Input Validation is very essential to make sure you don't have invalid data in your database.

In Chapter-2, we added the validation starter which adds the validation capabilities to our Spring Boot REST API.

Here are some validation rules we are going to implement:

- Expense ID Should not be null/empty, and it should only accept alphanumeric characters.
- Expense Name should not be null/empty, and it should only accept alphabets from a-z and A-Z.
- Expense Amount should accept only a minimum value of 0.

We have several annotations which will communicate our intent for validation. Eg:

- @NotNull as the name suggest makes sure that the value is Not Null
- @NotEmpty makes sure that the value is Not Empty.
- @NotBlank makes sure that the value is Not Empty and Not Null.
- Similar, we have @Pattern annotation which takes in Regular expression to validate the input.
- Finally, we have the @Min annotation, which makes sure that the provided value falls above the minimum range.

Our `ExpenseDto.java` class looks like below, after adding the necessary annotations for the validation.

```java
package com.programming.techie.expensetracker.dto;

import com.programming.techie.expensetracker.model.ExpenseCategory;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExpenseDto {
    @NotBlank
    @Pattern(regexp = "[A-Za-z0-9]+")
    private String id;
    @NotBlank
    @Pattern(regexp = "[a-zA-Z]")
    private String expenseName;
    private ExpenseCategory expenseCategory;
    @Min(value = 0)
    private BigDecimal expenseAmount;
}
```

Perform Manual Tests on our API

So we finished the basic implementation of our REST API, now let's go ahead and start the `ExpenseTrackerApplication` and test out our API using an HTTP Client.

You can use any HTTP Client to perform tests, I am using Postman, but you can use any HTTP Client you like.

To run the application, type the below command, or start the `ExpenseTrackerApplication` class in your favourite IDE.

If you are using Linux/macOS

```
./mvnw spring-boot:run
```

If you are using Windows

```
./mvnw.cmd spring-boot:run
```

Testing `POST` Requests

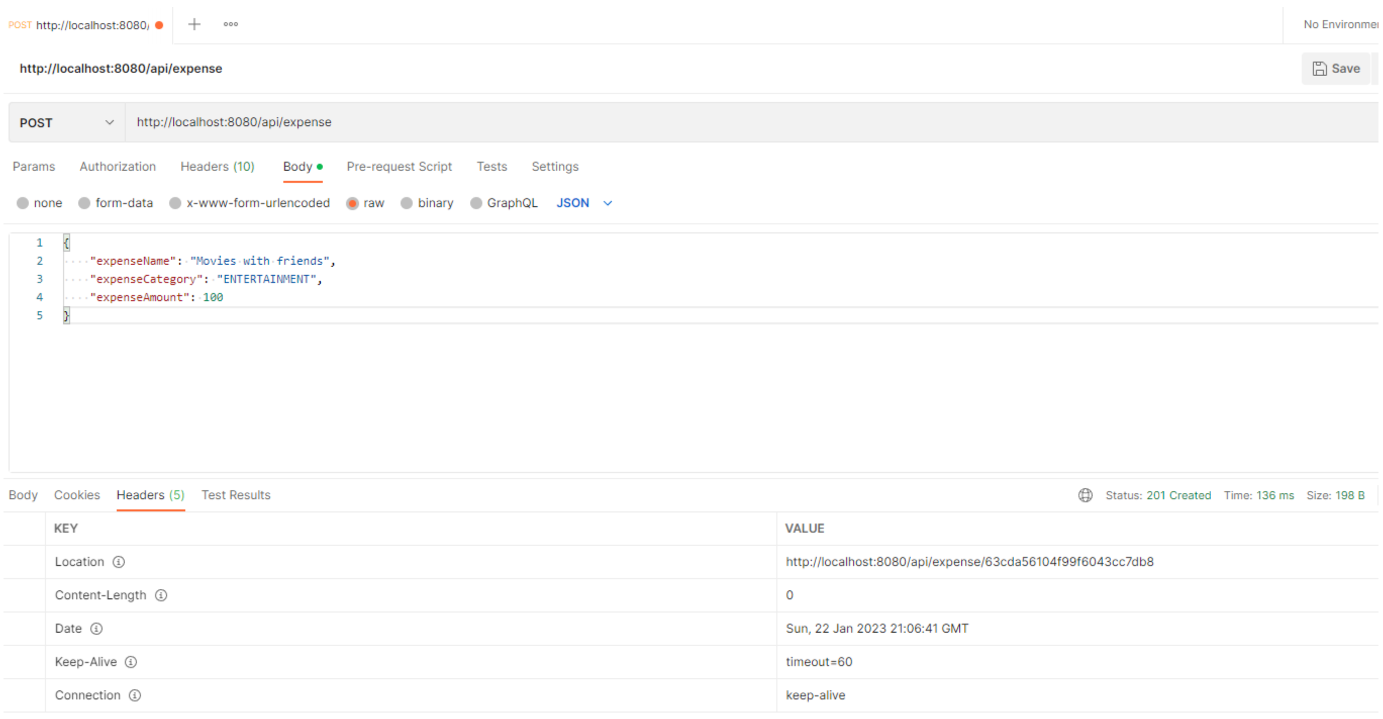
As we don't have any existing data in our application, let's start our test by creating some expenses.

I am going to send a `POST` request with the below Request Body:

```
{
    "expenseName": "Movies with friends",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 100
}
```

to the URL: `http:localhost:8080/api/expense`

This request should return an empty response body with HTTP Status as `201 Created`, you should also observe that in the `Location` header we can also see the URL to access the created `Expense`.


POST http://localhost:8080/api/expense

```
{
    "expenseName": "Movies with friends",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 100
}
```

Status: 201 Created Time: 136 ms Size: 198 B

KEY	VALUE
Location	http://localhost:8080/api/expense/63cda56104f99f6043cc7db8
Content-Length	0
Date	Sun, 22 Jan 2023 21:06:41 GMT
Keep-Alive	timeout=60
Connection	keep-alive

Testing GET Requests

So now let's test the GET requests by calling the two available endpoints:

- The Get All Expenses Endpoint
- The Get Expense Endpoint which takes in the Expense Id as path variable.

Testing Get All Expense Endpoint

Let's make a HTTP GET request to the URL - http://localhost:8080/api/expense

It should return the `Expense` we created before.

You can also create some other expenses using the above `POST` endpoint to test, whether the endpoint is indeed returning all `Expense` objects or not.

Testing Get Expense Endpoint

For this, we already have the URL, from the `Location` header when we made the `POST` request, just copy the URL and paste it in your HTTP Client.

You should see be able to see the given expense.

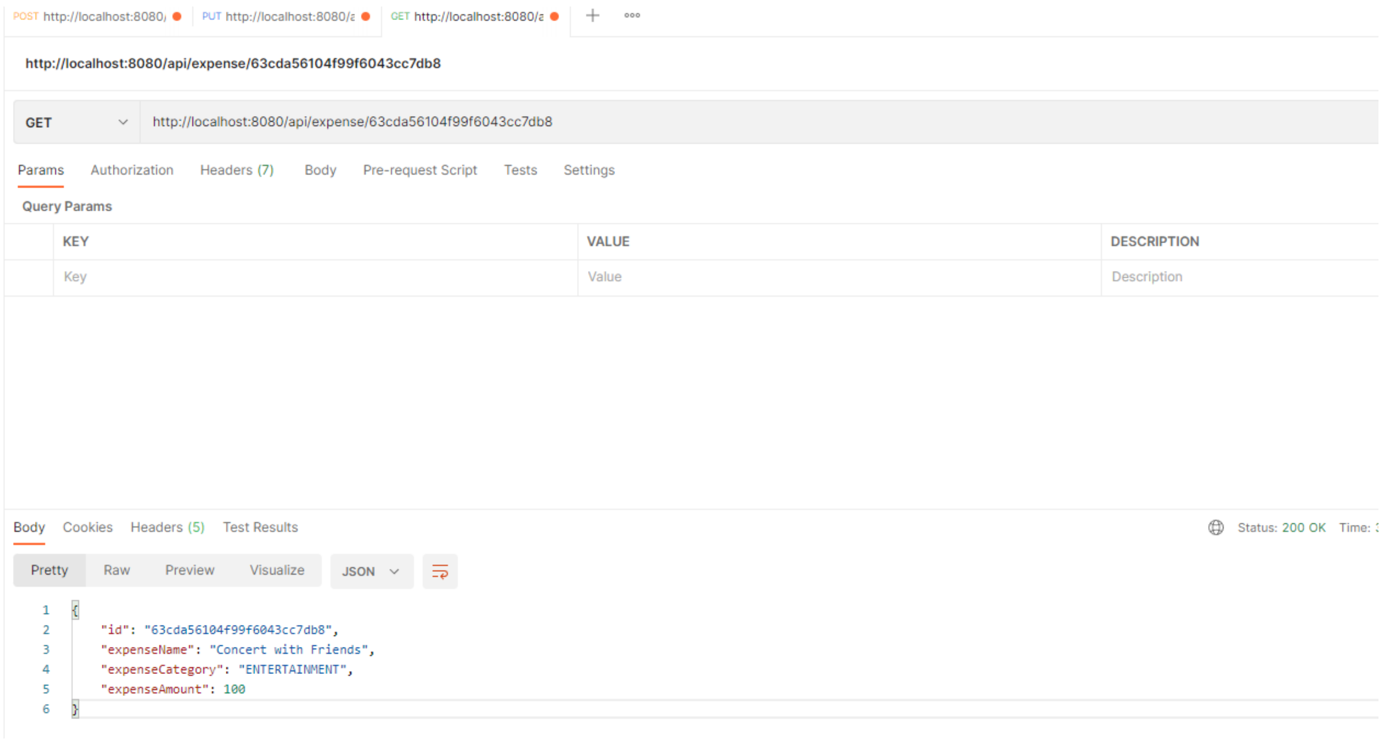
Testing PUT Request

Now let's try to update the name of the expense from "Movies with Friends" to "Concert with Friends", as we are making a PUT request, we have to send the whole Request Body again, to make sure that only expenseName attribute is updated.

```
{
    "expenseName": "Concert with Friends",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 100
}
```

You should see an empty response with HTTP Status as 200 OK

Let's access the Get Expense Endpoint again and verify if we see the updated Expense name (or) not.



http://localhost:8080/api/expense/63cda56104f99f6043cc7db8

GET http://localhost:8080/api/expense/63cda56104f99f6043cc7db8

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time:

```
{
    "id": "63cda56104f99f6043cc7db8",
    "expenseName": "Concert with Friends",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 100
}
```

As you can see, the expenseName attribute is updated successfully.

Testing DELETE Request

Now let's go ahead and test the last and final endpoint - to delete an Expense.

We are going to make a request to the http://localhost:8080/api/expense/<your-expense-id>

And you should receive an empty response with HTTP Status as 204-No Content

You can also verify if the delete operation worked correctly (or) not by querying the Get Expense Endpoint (or) Get All Expenses Endpoint.

Summary

In this chapter, we covered a lot, starting with creating our Domain Model, installing and configuring MongoDB in our application, creating Repository, Service and Controller classes to manage our Expense objects.

Finally, we also performed some manual tests using `Postman`.

But we still don’t have Automated tests in our application, in the next chapter, we will learn how to write Automated Unit and Integration Tests for our Expense Tracker API.

PART FIVE

Write Automated Tests for our REST API

In the previous chapter, you learned how to build the REST API using Spring Boot, and we performed some manual tests using Postman HTTP Client.

In this chapter, you will learn how to write Automated Unit and Integration Tests for your API using Junit 5, Spring Boot Test Support and Test Containers.

TestContainers provide us throw away instance of external infrastructure like Databases, Message Queues, Web Browsers, etc. Instead of using embedded versions of this software, we can run a docker container and test the application. In this way, we will test our application closer to production setup.

Unit Testing REST API using @WebMvcTest

The Spring Test Framework which is part of the `spring-boot-starter-test` library provides us with an annotation called `@WebMvcTest` which is a specialized annotation that will create the Spring Context for us with only beans which are related to the Spring MVC components like `@Controller`, `@RestController`, `@AutoconfigureWebMvc` etc.

This is how the annotation looks like:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(WebMvcTestContextBootstrapper.class)
@ExtendWith({SpringExtension.class})
@OverrideAutoConfiguration(
    enabled = false
)
@TypeExcludeFilters({WebMvcTypeExcludeFilter.class})
@AutoConfigureCache
@AutoConfigureWebMvc
@AutoConfigureMockMvc
@ImportAutoConfiguration
public @interface WebMvcTest {
    ...
    ...
}
```

You can see that this annotation acts as a wrapper for many other annotations related to Spring MVC components, mainly you can observe the `@AutoconfigureMockMvc` annotations, which autoconfigures a Mocked Web Servlet Environment to run tests for our controllers.

Using `@WebMvcTest` annotation, you can specifically test the behaviour of your Controllers.

Here is how the test for `ExpenseController.java` class looks like, we create a class called

ExpenseControllerTest.java:

```
package com.programming.techie.expensetracker.web;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.ExpenseCategory;
import com.programming.techie.expensetracker.service.ExpenseService;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.HttpHeaders;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

import java.math.BigDecimal;
import java.util.Objects;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;

@WebMvcTest(controllers = ExpenseController.class) ①
class ExpenseControllerTest {

    @MockBean ②
    private ExpenseService expenseService;
    @Autowired ③
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;

    @Test
    @DisplayName("Should Create Expense") ④
    void shouldCreateExpense() throws Exception {
        ExpenseDto expenseDto = ExpenseDto.builder()
                .expenseCategory(ExpenseCategory.ENTERTAINMENT)
                .expenseName("Movies")
                .expenseAmount(BigDecimal.TEN)
                .build();

        Mockito.when(expenseService.addExpense(expenseDto)).thenReturn("123");
        MvcResult mvcResult = mockMvc.perform(post("/api/expense") ⑤
                        .content(objectMapper.writeValueAsString(expenseDto)))
                .andExpect(MockMvcResultMatchers.status().isCreated())
                .andExpect(MockMvcResultMatchers.header().exists(HttpHeaders.LOCATION))
                .andReturn();

        assertTrue(Objects.requireNonNull(mvcResult.getResponse().getHeaderValue
(HttpHeaders.LOCATION))
                .toString().contains("123"));

    }
}
```

① The `@WebMvcTest` annotation, takes the `ExpenseController.class` as the value for the `controllers` attribute. Spring Boot at the time of startup, will create the Application Context with the `ExpenseController` bean.

② As we want to test the `ExpenseController.class` in isolation, we are going to mock the `ExpenseService.java` bean, with the `@MockBean` annotation.

③ We are going to autowire the `MockMvc` bean, which is autoconfigured already through our `@WebMvcTest` annotation.

④ Now in the `shouldCreateExpense` test, we first prepared an `ExpenseDto` object, by providing the test data. And then when the test executes the `expenseService.addExpense()` method, we return the `expenseDto` object as response using Mockito.

⑤ Finally, we are going to make use of `MockMvc` to make the call to our required endpoint, ie. /api/expense, using the `mockMvc.perform` method. Notice that, we converted the `expenseDto` object into a string using the `ObjectMapper's `writeValueAsString` method. After that we verify if we receive an HTTP Status - 201 (CREATED) response or not. And then we assert if there is a `Location` Header present in the response or not, and assert the value of the header.

While importing the `ObjectMapper` class in to the class, make sure to import the one from `com.fasterxml.jackson.databind.ObjectMapper` instead of `org.testcontainers.shaded.com.fasterxml.jackson.databind.ObjectMapper;`

This is how we write unit tests for the Spring MVC Controllers. Now let's learn how to write tests Integration Tests by using `@SpringBootTest` annotation along with Test Containers.

Integration Testing with `@SpringBootTest`

We can also write Integration Tests, by using the `@SpringBootTest` annotation, this will spinup the whole application context while running the tests.

This is how the integration test looks like by using `TestContainers` to spinup the Mongodb.

```
package com.programming.techie.expensetracker;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.ExpenseCategory;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.test.context.DynamicPropertyRegistry;
import org.springframework.test.context.DynamicPropertySource;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import org.testcontainers.containers.MongoDBContainer;
import org.testcontainers.junit.jupiter.Container;

import java.math.BigDecimal;
import java.net.URI;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) ①
@AutoConfigureMockMvc ②
class ExpenseTrackerRestApiApplicationTests {

    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;
    @Container ③
    static MongoDBContainer mongoDBContainer = new MongoDBContainer("mongo:6.0.4");

    static {
        mongoDBContainer
                .start(); ④
    }

    @DynamicPropertySource ⑤
    static void overrideProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.data.mongodb.uri", mongoDBContainer::getReplicaSetUrl);
    }
    @Test
    @DisplayName("Should Create Expense, and verify the expense by GET") ⑥
    void shouldCreateExpenseAndGetTheExpense() throws Exception {
        ExpenseDto expenseDto = ExpenseDto.builder()
                .expenseCategory(ExpenseCategory.ENTERTAINMENT)
                .expenseName("Movies")
                .expenseAmount(BigDecimal.TEN)
                .build();

        MvcResult mvcResult = mockMvc.perform(post("/api/expense")
                        .contentType(MediaType.APPLICATION_JSON_VALUE)
                        .content(objectMapper.writeValueAsString(expenseDto)))
                .andExpect(MockMvcResultMatchers.status().isCreated())
                .andExpect(MockMvcResultMatchers.header().exists(HttpHeaders.LOCATION))
                .andReturn();

        String expenseUrl = mvcResult.getResponse().getHeaderValue(HttpHeaders.LOCATION
).toString();

        mockMvc.perform(get(new URI(expenseUrl))) ⑦
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.expenseAmount").value
(BigDecimal.TEN))
                .andExpect(MockMvcResultMatchers.jsonPath("$.expenseCategory").value
(ExpenseCategory.ENTERTAINMENT.toString()))
                .andExpect(MockMvcResultMatchers.jsonPath("$.expenseName").value(
"Movies"));

    }

}
```

① Here we defined the `@SpringBootTest` annotation, where we define the `webEnvironment` attribute as `SpringBootTest.WebEnvironment.RANDOM_PORT`, this will run the Spring Application Context on a random port, if we don't provide this property, it will try to start the context on the default port.

② Similar to the unit test using `@WebMvcTest` we are using `MockMvc` to perform calls to our Controller endpoints, hence we are using the `@AutoconfigureMockMvc` annotation and auto-wired the bean in the test class.

③ Here you can observe the `@Container` annotation, which will take care of starting the `MongoDbContainer`. During the test startup, Testcontainers will check if there is a Mongodb docker image on our machine, with version 6.0.4 and will download the image if it does not exist.

④ Inside the `static` block, we are starting the container.

⑤ We can dynamically override the `spring.data.mongodb.uri` property in our application using the `@DynamicPropertySource` annotation, here we can get the URL of the MongoDB inside the docker container using the `mongoDbContiner::getReplicaSetUrl()` method.

⑥ Finally, we have our main test, which constructs the required `expenseDto` object and calls our `/api/expense` endpoint, and we verify whether the expense is created our not.

⑦ We can verify the expense by using the URL which is returned inside the `Location` header. We make the call to the expense URL using the `mockMvc.perform()` and verify the JSON response by asserting the fields `expenseName`, `expenseAmount` and `expenseCategory`.

Summary

In this chapter, you learned how to write Unit and Integration Tests for your REST API and learned how to work with Testcontainers.

In the next chapter, we will learn how to document our REST APIs using `OpenAPI`.

PART SIX

Documenting REST API using OpenAPI 3.0

In this chapter, we are going to learn how to document our REST APIs.

Why should we document our REST APIs?

In the real world, consumers of an API should have a good understanding of the REST APIs they are using. Having good documentation is vital in helping the users to use the API effectively.

Having good documentation for our REST APIs is necessary. On the other hand, maintaining the documentation manually is tiresome and error-prone.

Generating REST API Documentation using OpenAPI

The Springdoc Openapi project makes this process of generating REST API documentation quick and painless. Using these tools, we can automate the process of documentation.

Specifically, the `springdoc-openapi` library helps to automate the generation of API documentation by scanning application at runtime to infer API semantics based on spring configurations, class structure and various annotations.

It uses Swagger under the hood to expose a Web UI where we can browse through the endpoints defined in our Spring Boot Application.

What is Swagger?

So what is Swagger? It is an OPEN API specification that is created as a standard to describe your REST API.

Adding `springdoc-openapi` Dependencies to project

Inside our `pom.xml` file, add the following maven dependencies. This should download the required `springdoc-openapi` dependencies to our project.

```
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.0.2</version>
        </dependency>
```

- The `springdoc-openapi-starter-webmvc-ui` dependency adds the swagger-ui webjar files to our project, and provides us a Swagger based UI to view the REST API documentation. You can see how it looks like in the next section.

```
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-api</artifactId>
            <version>2.0.2</version>
        </dependency>
```

- If you just want to expose the API Documentation as an API response itself, instead of the UI, then you can use the `springdoc-openapi-starter-webmvc-api` dependency.

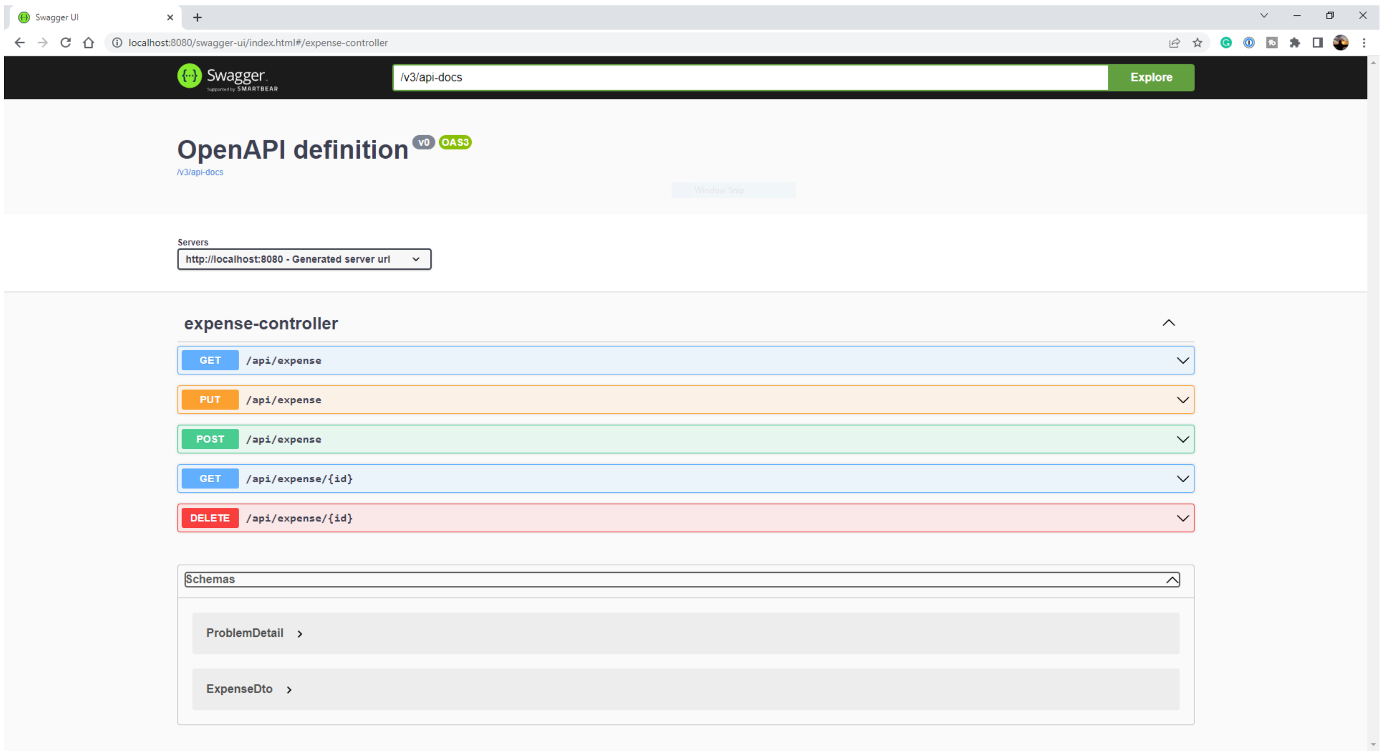
Configure Swagger UI Endpoints

After adding the above dependency, we can configure the Swagger UI Endpoint, by adding the below property:

```
springdoc.swagger-ui.path=/swagger-ui.html
```

By adding this property, you can find the REST API Documentation UI at the link - http://localhost:8080/swagger-ui.html

This is how the documentation UI looks like:

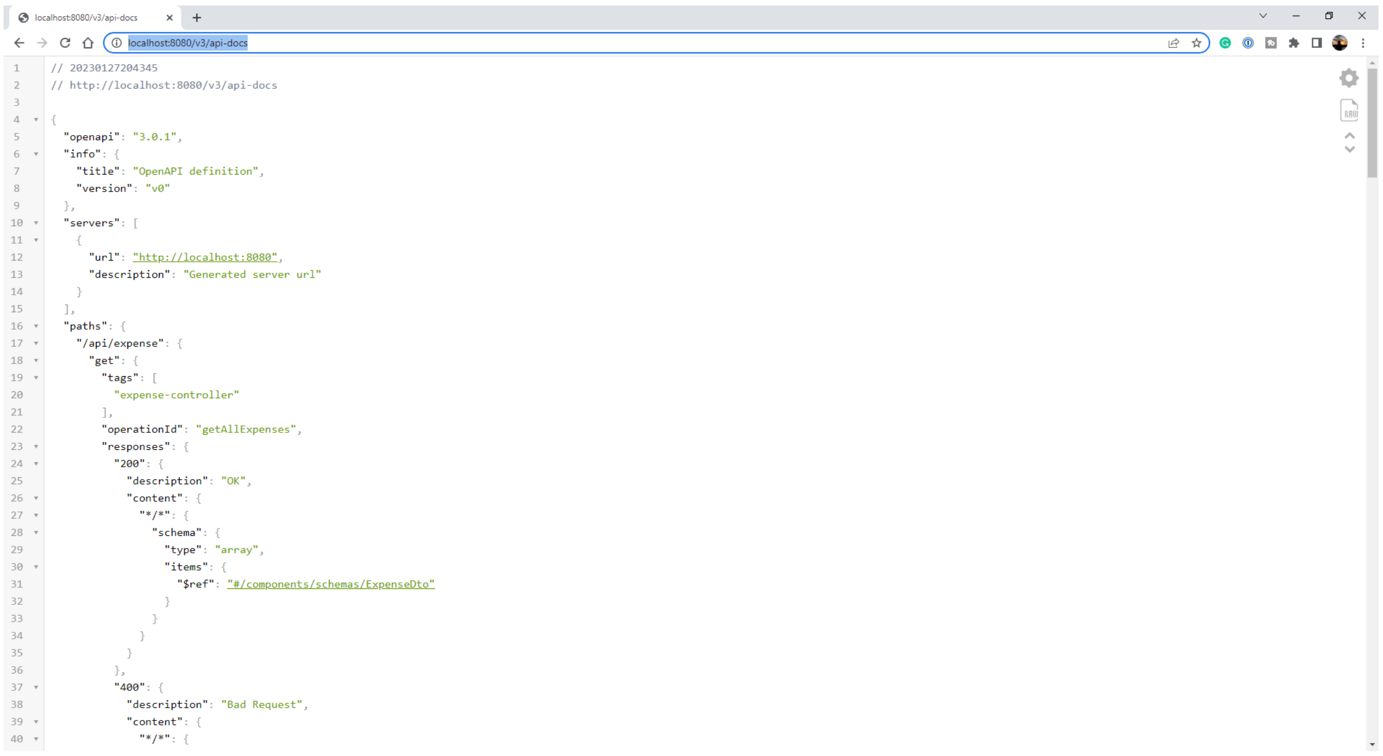


Configure Swagger Docs API Endpoints

We can configure the Swagger API Endpoint, by adding the below property:

```
springdoc.api-docs.path=/api-docs
```

By adding this property, you can find the REST API Documentation at the link - http://localhost:8080/api-docs



```
// 20230127204345
// http://localhost:8080/v3/api-docs

{
  "openapi": "3.0.1",
  "info": {
    "title": "OpenAPI definition",
    "version": "v0"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Generated server url"
    }
  ],
  "paths": {
    "/api/expense": {
      "get": {
        "tags": [
          "expense-controller"
        ],
        "operationId": "getAllExpenses",
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "*/*": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/ExpenseDto"
                  }
                }
              }
            }
          },
          "400": {
            "description": "Bad Request",
            "content": {
              "*/*": {
```

Customizing the Swagger Properties

If you observe the Swagger UI carefully, we see that the heading says - OpenAPI definition which is a bit generic and out of the box text from the library.

In a real world setup, you want to change this according to your application/API.

We can customize this properties by defining a custom bean in our spring boot application. For that let's create a class called `OpenApiConfiguration.java` inside a package called `config`.

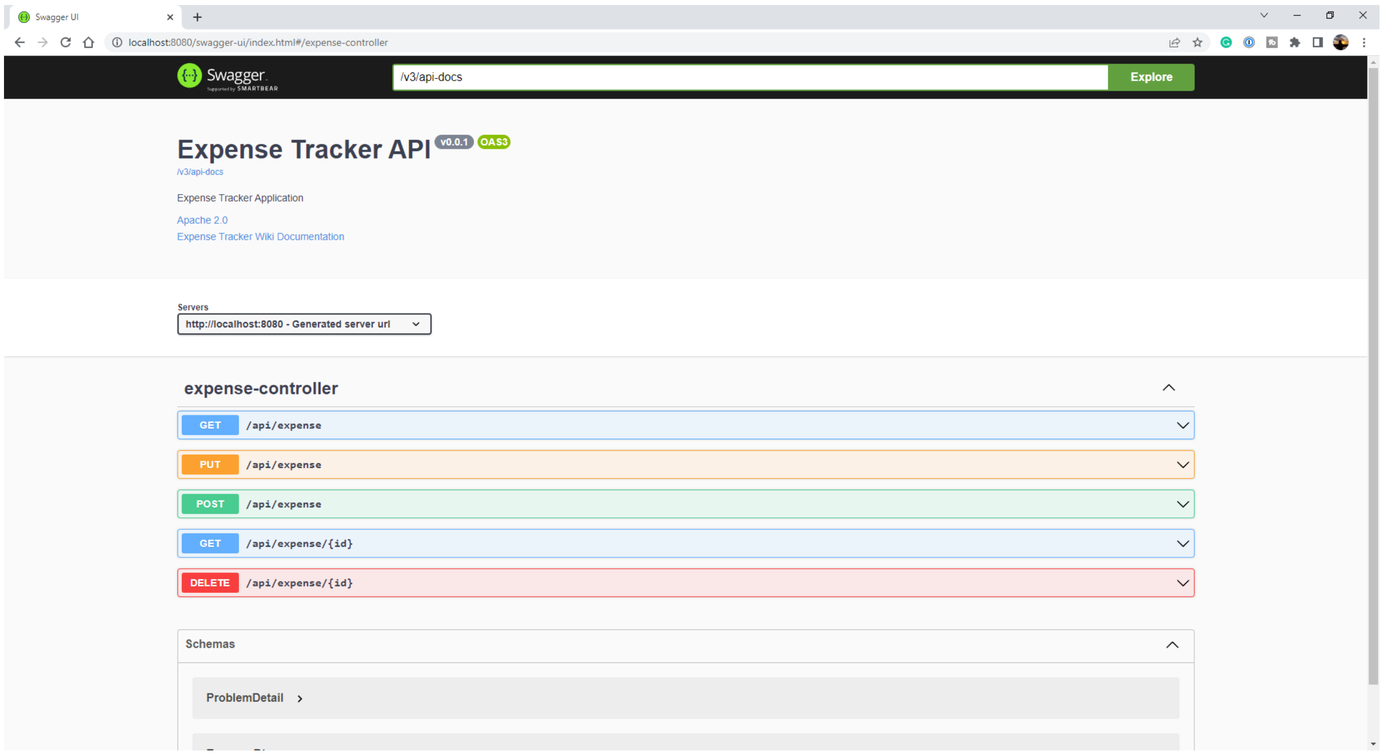
```
package com.programming.techie.expensetracker.config;

import io.swagger.v3.oas.models.ExternalDocumentation;
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;
import io.swagger.v3.oas.models.info.License;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OpenAPIConfiguration {

    @Bean
    public OpenAPI expenseAPI() {
        return new OpenAPI()
                .info(new Info().title("Expense Tracker API")
                        .description("Expense Tracker Application")
                        .version("v0.0.1")
                        .license(new License().name("Apache 2.0").url("https://expense-
tracker.org")))
                .externalDocs(new ExternalDocumentation()
                        .description("Expense Tracker Wiki Documentation")
                        .url("https://expensetracker.wiki/docs"));
    }
}
```

In the above class, we defined a bean by name `expenseAPI`, which creates an object of class `OpenApi`, which consumes all the necessary information about our application.

You can define various attributes like:

- The Title of the Application / API
- Description
- Version
- License information
- Links to external documentation along with URL

After adding this bean, we can simply restart our Spring Boot Application and when you open the link http://localhost:8080/api-docs you can see the custom information in our Documentation:



Bean Validation Support in Swagger UI

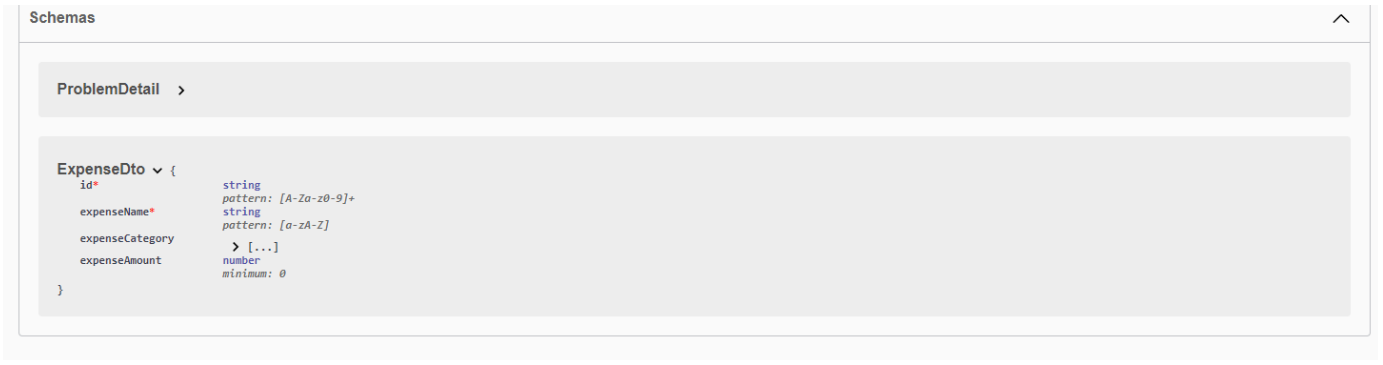
In the Chapter-4, we added Validation capabilities to our REST API, the `springdocs-openapi` library automatically detects this and displays it in our documentation.

For example, this is how our `ExpenseDto.java` class looks like:

```java
package com.programming.techie.expensetracker.dto;

import com.programming.techie.expensetracker.model.ExpenseCategory;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExpenseDto {
    @NotBlank
    @Pattern(regexp = "[A-Za-z0-9]+")
    private String id;
    @NotBlank
    @Pattern(regexp = "[a-zA-Z]")
    private String expenseName;
    private ExpenseCategory expenseCategory;
    @Min(value = 0)
    private BigDecimal expenseAmount;
}
```

You can observe that we are using multiple annotations like: * `@NotBlank` * `@Pattern` * `@Min`

In the Swagger UI, if we observe the `Schemas` section, and expand the ExpenseDto section, you can see that the fields are marked as mandatory, and the required Pattern and minimum value information is populated automatically. Refer to the image below:



```
ExpenseDto {
    id*                 string
                        pattern: [A-Za-z0-9]+
    expenseName*        string
                        pattern: [a-zA-Z]
    expenseCategory
                         > [...]
    expenseAmount       number
                        minimum: 0
}
```

Adding Custom Documentation for our Endpoints

If you expand any of the GET, POST, PUT, DELETE sections of the endpoints, you can observe that there is no description about the API, in the real world, it's essential to document the behaviour of the endpoints. After all, that's the whole reason to create the documentation :D

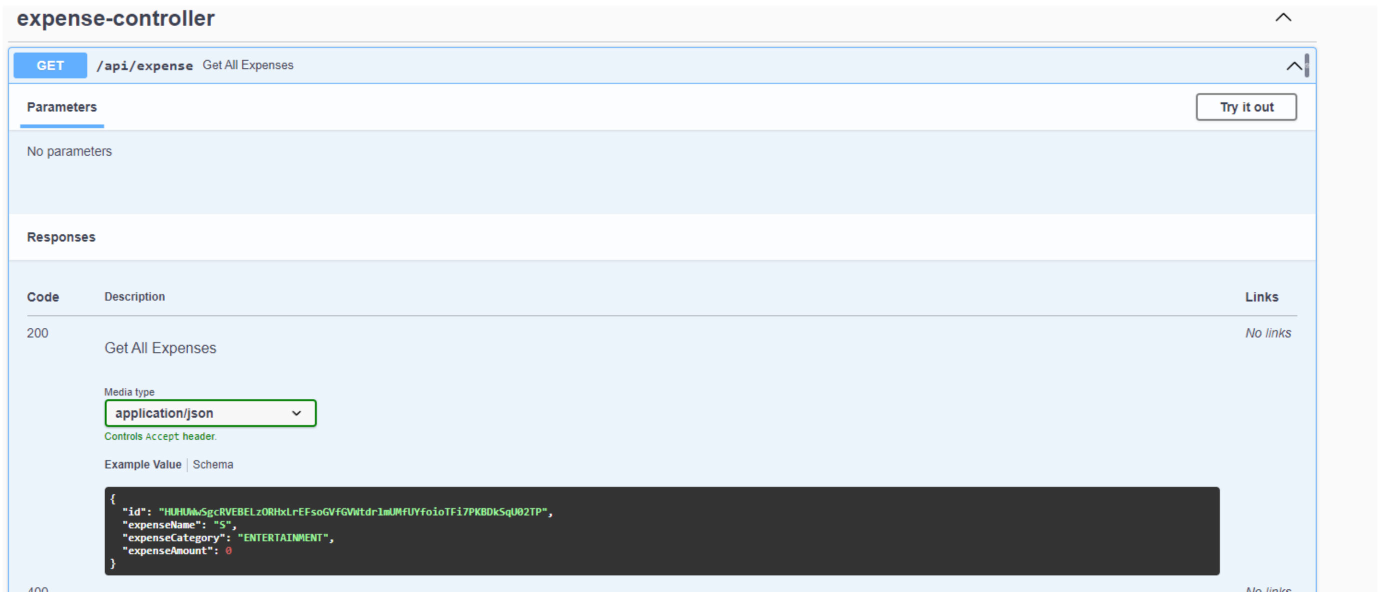
We can customize the description of the endpoints with the help of several annotations from the `springdocs-openapi` library using the `@Operation`, `@ApiResponses` and `@ApiResponse` annotations.

Here is a complete example of how `ExpenseController.java` class looks like after adding the above annotations:

```
package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.service.ExpenseService;
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.media.Content;
import io.swagger.v3.oas.annotations.media.Schema;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.responses.ApiResponses;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/expense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

    @Operation(summary = "Add an Expense")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "201", description = "Expense Created",
                    content = {@Content(mediaType = "application/json",
                            schema = @Schema(implementation = ExpenseDto.class))}),
            @ApiResponse(responseCode = "400", description = "Invalid/Bad Request",
                    content = @Content)})
    @PostMapping
    public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
        return ResponseEntity.created(location)
                .build();
    }

    @PutMapping
    @ResponseStatus(HttpStatus.OK)
    @Operation(summary = "Update an Expense")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Expense Updated",
                    content = {@Content(mediaType = "application/json",
                            schema = @Schema(implementation = ExpenseDto.class))}),
            @ApiResponse(responseCode = "400", description = "Invalid/Bad Request",
                    content = @Content),
            @ApiResponse(responseCode = "404", description = "Expense Not Found",
                    content = @Content)})
    public void updateExpense(@RequestBody ExpenseDto expense) {
        expenseService.updateExpense(expense);
    }

    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    @Operation(summary = "Get All Expenses")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Get All Expenses",
                    content = {@Content(mediaType = "application/json",
                            schema = @Schema(implementation = ExpenseDto.class))})})
    public List<ExpenseDto> getAllExpenses() {
        return expenseService.getAllExpenses();
    }

    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    @Operation(summary = "Get an Expense")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Get Single Expense",
                    content = {@Content(mediaType = "application/json",
                            schema = @Schema(implementation = ExpenseDto.class))}),
            @ApiResponse(responseCode = "404", description = "Expense Not Found",
                    content = @Content)})
    public ExpenseDto getExpense(@PathVariable String id) {
        return expenseService.getExpense(id);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @Operation(summary = "Delete an Expense")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "204", description = "Expense Deleted",
                    content = {@Content(mediaType = "application/json",
                            schema = @Schema(implementation = ExpenseDto.class))}),
            @ApiResponse(responseCode = "404", description = "Expense Not Found",
                    content = @Content)})
    public void deleteExpense(@PathVariable String id) {
        expenseService.deleteExpense(id);
    }

}
```

As you can see, the `@Operation` annotation is used to provide a small description of the API and the `@ApiResponses` annotation which takes in a list of `@ApiResponse` values, is used to define the response code as well as additional description about the response code.

After adding the above annotations just restart the application, and you can see the description of the Endpoints in the Swagger UI documentations



```
{
  "id": "HUHUWwSgcRVEBELzORHxLrEFsoGVfGVWtdrlmUMfUYfoioTFi7PKBDkSqU02TP",
  "expenseName": "S",
  "expenseCategory": "ENTERTAINMENT",
  "expenseAmount": 0
}
```

Summary

In this chapter, you learned the importance of documenting your REST API, why we need to document our REST APIs, and then implemented the documentation using SpringDocs OpenAPI library.

In the next chapter, we will learn how to handle exceptions in our REST API.

PART SEVEN

Exception Handling in REST APIs

In this chapter, you will learn how to handle exceptions thrown by our ExpenseTracker application.

In the `ExpensService` class, inside the `getExpense` method, we are reading the expense from the database by id, and if there is no expense with the given id, then we are throwing an `ExpenseNotFoundException`.

This is how it looks like when you make a call to the Get Expense Endpoint - /api/expense/{expenseId} with invalid `expenseId`:

```
{
"timestamp": "2023-01-27T18:02:22.324+00:00",
"status": 500,
"error": "Internal Server Error",
"path": "/api/expense/1cda56104f99f6043cc7db8"
}
```

The above error is fine, but in the real-world you may want to add much more details when an exception is raised in your application.

For example:

- You may want to see the exception message as part of the response.
- You may want to add a custom exception handling URL, so that the client can redirect to that particular URL.
- Categorize the Exception into different formats like: Generic, Technical exceptions.

Spring Boot 3 includes a library from Zalando called problem-spring-web, and provides the functionality using the `ProblemDetail` class.

Define Handlers using @ExceptionHandler annotation

We can define an Exception Handler for the `ExpenseNotFoundException` by first defining a class called `ExpenseExceptionHandler` that has the `@ControllerAdvice` annotation and extends the `ResponseEntityExceptionHandler` class.

The `@ControllerAdvice` annotation is used to execute a common code whenever an event occurs inside our Controllers.

As we are dealing with creating custom responses by handling the exceptions thrown by our Expense Tracker application, we are extending the class - `ResponseEntityExceptionHandler`.

The complete code for the `ExpenseExceptionHandler` can be found below:

```java
package com.programming.techie.expensetracker.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

import java.net.URI;
import java.time.Instant;

@ControllerAdvice
public class ExpenseExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ExpenseNotFoundException.class)
    ProblemDetail handleExpenseNotFoundException(ExpenseNotFoundException e) {
        ProblemDetail problemDetail = ProblemDetail.forStatusAndDetail(HttpStatus
.NOT_FOUND, e.getMessage());
        problemDetail.setTitle("Expense Not Found");
        problemDetail.setType(URI.create("https://api.expenses.com/errors/not-found"));
        problemDetail.setProperty("errorCategory", "Generic");
        problemDetail.setProperty("timestamp", Instant.now());
        return problemDetail;
    }
}
```

Here inside the `handleExpenseNotFoundException` method, we added the `@ExceptionHandler` annotation on top of the method and inside the method, we created an object of `ProblemDetail` class, and we are providing the necessary details like `title`, `type`, and some custom properties like `errorCategory`, `timestamp`, etc. We are also setting the HTTP Status as 404 (NOT_FOUND).

After adding this class, if you re-start the application and call the Get Expense Endpoint, with an invalid `expenseId` you should see the below Error response:

```json
{
    "type": "https://api.expenses.com/errors/not-found",
    "title": "Expense Not Found",
    "status": 404,
    "detail": "Cannot Find Expense by ID - 1cda56104f99f6043cc7db8",
    "instance": "/api/expense/1cda56104f99f6043cc7db8",
    "errorCategory": "Generic",
    "timestamp": "2023-01-27T18:10:48.347319800Z"
}
```

You can observe that this response is much more clear than the default error response generated by Spring Boot.

You can customize this `ProblemDetail` object and add any information you feel is necessary for your APIs.

Adapting our tests to handle the Exceptions

As we made some changes in our exception handling logic, now let's go ahead and write a test to verify that the exception handling behavior.

Inside the `ExpenseController.java`, add the below test method:

```
@WebMvcTest(controllers = ExpenseController.class)
class ExpenseControllerTest {

    ...
    ...

    @Test
    @DisplayName("Should Return 404 Not Found Exception when calling expense endpoint
with invalid id")
    void shouldReturn404ErrorResponseForGETWithInvalidId() throws Exception {
        Mockito.when(expenseService.getExpense("123")).thenThrow(new
ExpenseNotFoundException("Cannot find Expense By id - 123")); ①

        MvcResult mvcResult = mockMvc.perform(get("/api/expense/123") ②
                        .contentType(MediaType.APPLICATION_JSON_VALUE))
                .andExpect(MockMvcResultMatchers.status().isNotFound())
                .andExpect(MockMvcResultMatchers.jsonPath("$.type").value
("https://api.expenses.com/errors/not-found"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.title").value("Expense Not
Found"))
                .andExpect(MockMvcResultMatchers.jsonPath("$.errorCategory").value
("Generic"))
                .andReturn();

    }
}
```

① Inside the test, you can observe that we are simulating the error using Mockito, by making sure that our `expenseService.getExpense()` method throws an `ExpenseNotFoundException`

② Now using the `mockMvc.perform()` method we are making a GET call to the `/api/expense/123` endpoint, and first verifying that the response status is Not Found, later we are verifying the fields we added inside the `ExpenseExceptionHandler` class like `title`, `type`, `errorCategory`, etc.

Summary

In this final chapter, you learned how to handle custom exceptions in your Spring Boot REST APIs.

About the author

Sai Subramanyam Upadhyayula is a Software Engineer from India, working in the Netherlands for a Swiss Telecom Giant called Swisscom.

In his professional experience, he worked in several roles across the Software Engineering Lifecycles and worked with several technologies like Java, Spring Boot, Spring Cloud, Typescript, Angular, HTML, CSS, Java Script, Golang and Devops Tooling.

He is a published author of the book - "Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices", you can read the book here if you are interested - https://link.springer.com/book/10.1007/978-1-4842-8792-7

You can follow him at:

- Twitter - https://twitter.com/sai90_u
- Github - https://github.com/SaiUpadhyayula
- Linkedin - https://www.linkedin.com/in/sai-upadhyayula/